Tutorial Four: Linear Regression

Imad Pasha
Chris Agostino

February 17, 2017

1 Introduction

When looking at the results of experiments, it is critically important to be able to fit curves to scattered data points. We will demonstrate that doing so in python is relatively simple, but the theory behind how it works is a bit more involved. For the upper division lab, you may be expected to create best fit lines from scratch, so we are going to briefly go over that here.

2 Linear Least Squares

There are different ways of fitting curves to scattered points. One of the most frequently used is known as Linear Least Squares, a subset of Bayesian generalized fitting. Note, we can fit any order polynomial, not just straight lines, using this method. The "linear" part refers to how the distance between the data point and the line is measured, as we describe momentarily. The method of LLS fits a line to your data that minimizes the squared distances between all the points and the line. The reason for choosing the squared distances is that some points will lie below your line, but distances are positive. By squaring we allow for points below the line to also be a "positive" distance away from the line. The formula for generating a LLS fit outputs the constants of the equation $(a0 + a_1x + a_2x^2 + ...)$ for as many orders as you require based on the degree/order of your fit. For the linear case, then, LLS outputs a slope and a y-intercept. The formula requires linear algebra, which some of you may not yet have taken, and looks like this:

$$\begin{pmatrix} N & \sum x_i & \sum x_i^2 & \cdots & \sum x_i^m \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \cdots & x_i^{m+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \sum x_i^m & \sum x_i^{m+1} & \sum x_i^{m+2} & \cdots & \sum x_i^{2m} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} \sum y_i \\ \sum x_i y_i \\ \vdots \\ \sum x^{n-1} y_i \end{pmatrix} \tag{1}$$

This may look a little scary, but its not hard to implement. N is the number of data points you are trying to fit to. To enter the x sums, simply take your x array (such as an array of centroids), and run np.sum on them (squared, cubed, etc as required). The y_i are the y values of the data points, which can be multiplied by the x arrays within the np.sum function.

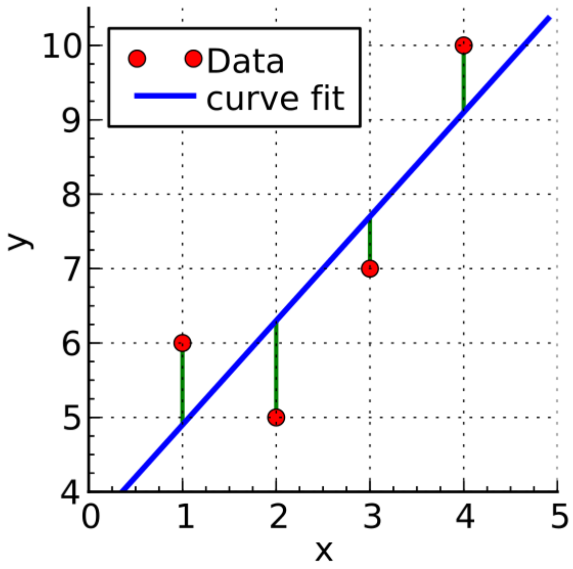

Figure 1: A linear least squares fit. Data are in red, fit is in blue, and residuals are in green. The method minimizes the residuals as much as possible.

3 Fitting a Straight Line

Equation 1 shows us how to fit any order polynomial to a set of data (based on how large you make the array). We are going to practice simply fitting an order 1 polynomial (a straight line) to some data. In this case, the LLS formula simplifies to

$$\begin{pmatrix} N & \sum x_i \\ \sum x_i & \sum x_i^2 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} \sum y_i \\ \sum x_i y_i \end{pmatrix} \tag{2}$$

where we are solving for c_1 and c_2, the slope and intercept of our best fit. We know N (it's just the number of data points), we know the x_i and y_i, so now it's just a matter of figuring out how to do matrix multiplication in python. If we remember (or crash course) our linear algebra, to get the c_1, c_2 on it's own, we need to multiply both sides *on the left* by the *inverse* of the (N...) array. So we arrive at:

$$\begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} N & \sum x_i \\ \sum x_i & \sum x_i^2 \end{pmatrix}^{-1} \cdot \begin{pmatrix} \sum y_i \\ \sum x_i y_i \end{pmatrix} \tag{3}$$

where the inverse of the (N...) matrix is now being dotted into the ($\sum y_i$...) array. The functions you'll find useful when setting this up:

- np.linalg.inv(arr)

- np.dot(arr1, arr2)

Remember that to set up multidimensional arrays, the format is np.array([[a,b,c,...],[d,e,f,g,...],...]) that is, lists nested inside a list nested inside the function call.

4 The Data

4.1 A function for loading the data

Located in the hw4 directory is a file called data.txt, which contains two columns: x and y. We don't really care what these represent at this point. Go ahead and use the loadtxt methods we've been using in the class to get th data file into python and the x values into an x-array and the y values into a y-array. The skeleton file we have provided you (hw4.py) has a function in it called load_data(data_file). Examine it to figure out how it wants you to set up the data to return it, and then call your function in your script to set the variables appropriately.

Once you've done that, go ahead and plot the data to see what we're working with. You'll want to use a symbol demarcation like '+' or 'o' . Here's what it should look like (try to get some practice by setting the axis labels, title, and creating a legend).

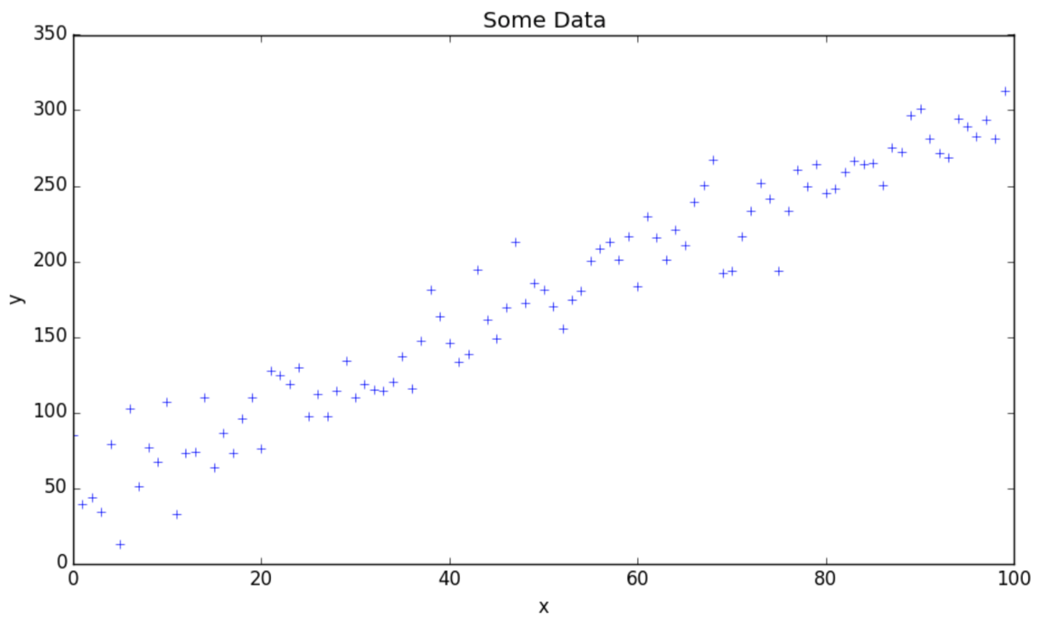

Figure 2: The data we will be fitting.

4.2 A function for fitting the data

Once you have the data in and plotted, you can disable your plotting command and turn to the second predefined function in the file: linear_fit. In the arguments and documentation, we've specified what types of data should come in, and in the return statement and documentation we've told you what should come out (namely, the fit coefficients). Within this function, you will be performing the Matrix math discussed in §2.

4.3 A function to plot the fit

This part is easy. The function plot_fit takes in your x-data and y-data. The first step in the function is to fit the data using the function you just wrote above and extract the coefficients. Next, you have to construct an actual line to plot over your data. The easiest (and required) way to do this is to use the x-array you already have, and construct a new array (something like y_fit) which evaluates your fit at every value of x in the original array. Next, you'll have some plotting statements to plot the scattered data and the fit you found as a straight line over the points. Finally, return the original x-array, and the new straight-line-fit y-array (comma separated).

You might want to perform your fit and use those values without a plot coming up every time (actually, you'll probably want to do this in the next section). In the arguments for your plot_fit function, add an argument called do_plot and set it to False, which means by default that variable will be False unless the user inputs something different. EX:

```
def plot_fit(x,y,do_plot=False):
```

then inside your function, set off your plotting commands inside an if-statement that checks the value of do_plot before deciding whether to actually plot the fit or not.

4.4 A function to plot the Residuals

When we are trying to evaluate whether a fit is good/sufficient, the first, and easiest thing to do is look at the residuals. We've plotted quantities similar to residuals in the past, so I wont harp on it here. Check out the function find_residuals to see what it should take as inputs and what it should return as outputs. It's also got a plot option defaulted to false. However, we are going to return more than just the array of the residuals, we're gonna do some basic statistics on them and return those as well. Within your function, calculate the mean of the residual array, the standard deviation of the residual array, and the maximum and minimum of the residual array. Stick these values into a dictionary with 'mean', 'std', 'min', and 'max' as keys, and the appropriate calculated numbers (you can actually set values to a dict via indexing!). At the end, you'll see that you return both the residual array itself as well as the dictionary with the stats about our residuals.

Run your function with plotting enabled, and take a look at your residuals. They should be relatively small (within 60 of 0), and randomly spread to both positive and negative values, like in the figure here. Now that you've gone through the pain of creating your own linear fitter, you'll probably remember that in previous tutorials, I simply called np.polyfit(x,y,degree) to generate fits. From now on, you should probably use that, instead of mucking through the matrix algebra yourself. But in Astro 120, you'll be asked to use the matrix version at least once, so it's good to understand how it's working.

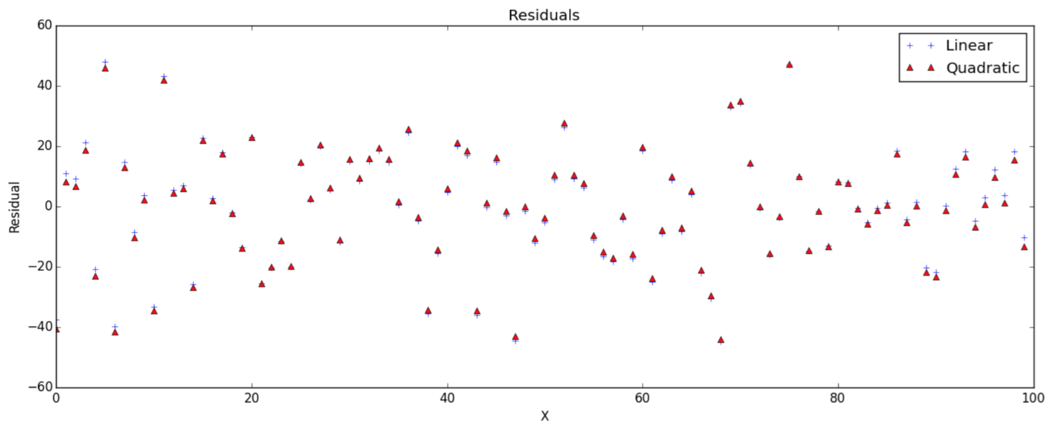

Figure 3: Residuals. I've plotted both the linear residuals as well as those created by fitting a quadratic. We can see that they lie almost on top of each other. This indicates that the data were probablyn inherently linear (they were), as a quadratic doesn't improve on our fit significantly.

5 Turning in the HW

Each of the functions you've been working with above has an associated okpy test associated with it, much like the notebooks we've been working with before. To run the tests on all your functions, you can navigate to the folder with your code in it and from the python terminal type

```
python ok
```

Making sure that you are in your pydecal environment. The first time you do this, it should ask for your bcourses email for a login. To see the outputs of each test, run the command with the verbose tag

```
python ok -v
```

And you can test locally without accessing the server by using python ok –local

You can test individual questions (they are numbered in order) using

```
python ok -q ‘q01’
```

etc. Once you are satisfied all is well (or even before then), you can submit using

```
python ok –submit
```